2801ICT Assignment 3

Peter Leddiman s5169483

Overview

Given an arbitrary weighted directed graph with non-negative weights, find the k-shortest loopless paths from any given source vertex to any given destination vertex.

Algorithm

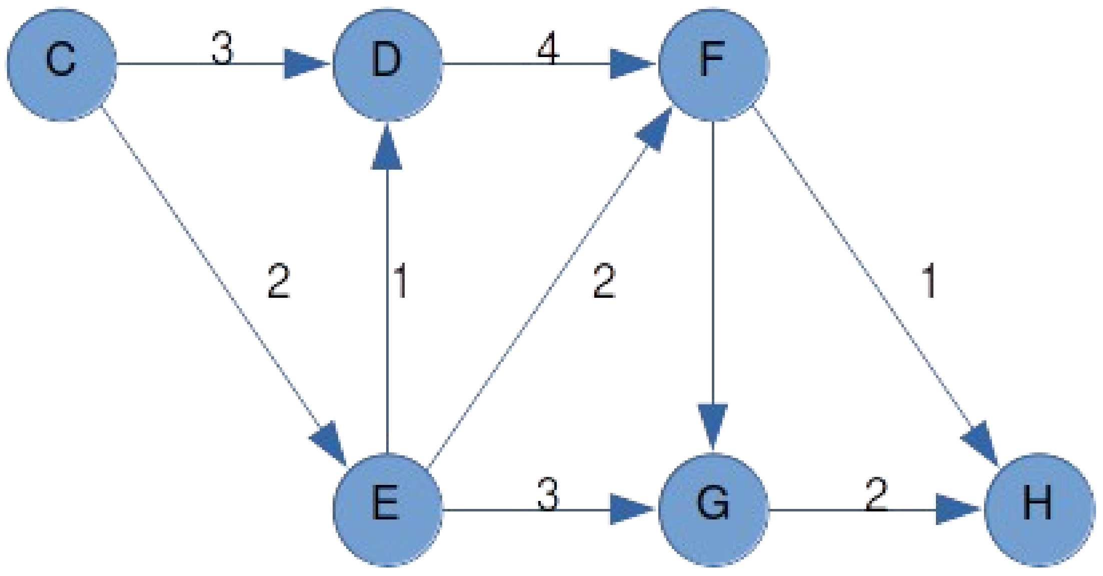
This problem can be easily solved using an A*-search from the source vertex to the destination. Typically an A*-search would finish once any path to the goal has been found, but we can simply continue the search as normal to find k paths in total. Doing this will find the k-shortest paths to the goal in order.

The only remaining difficulty is to construct a suitable heuristic for the A*-search. Recall that the heuristic in an A*-search is an estimate of the cost from the current node to a goal state. This heuristic must satisfy a few constraints; namely, that it must not overestimate the cost (but underestimation is fine), and that the costs must satisfy the triangle inequality; that is, if the heuristic for node A is h(A) and for node B it is h(B) and the cost to move from A to B is c(A, B), it must be that h(A) <= c(A, B) + h(B).

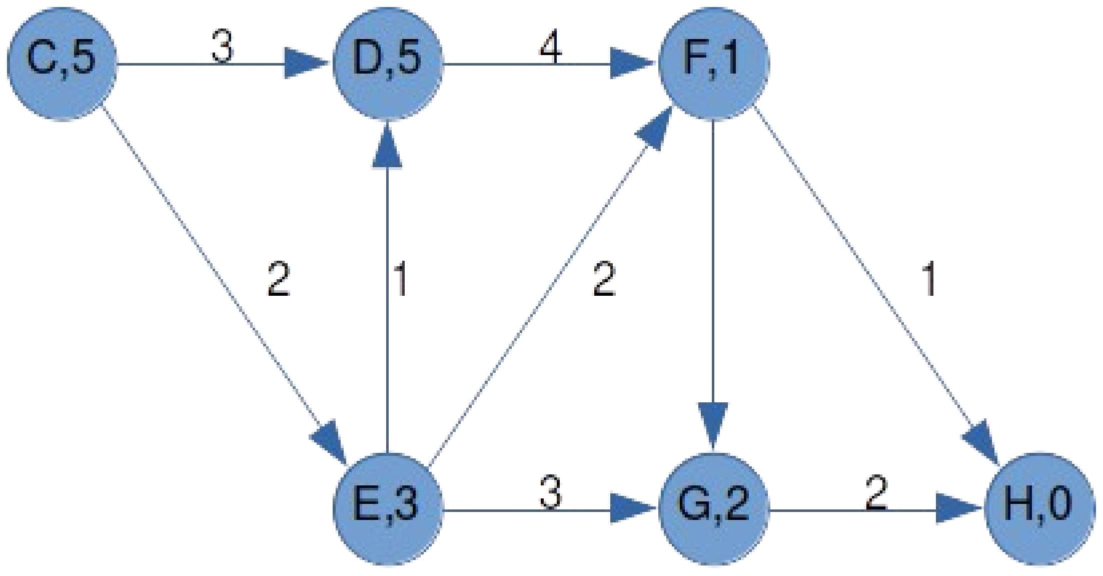
The performance of an A*-search varies greatly depending on the quality of the heuristic. In the worst case we could use the heuristic h(x) = 0 for all x, in which case the A*-search degenerates into a variant of Dijkstra's algorithm, taking O(E log E) time. If the heuristic is not an approximation but is in fact exact this reduces to O(V log E). This is because each time we add an element to the priority queue its priority will be the addition of the cost to reach that node and the cost to reach the destination. Therefore the nodes will be popped off the queue in the order that they appear along the shortest path.

This of course requires us to calculate the cost to reach the destination from every vertex in the graph. This can easily be achieved by performing Dijkstra's algorithm on the reversed graph; that is, reversing every directed edge and performing Dijkstra's algorithm starting at the destination.

So, given the following graph:



We first compute Dijkstra's algorithm in reverse to find the shortest path from every node to the destination:



We can then perform an A*-search very efficiently using the calculated shortest path cost as the heuristic.

The graph is stored as an adjacency list; that is, we store a list of vertices, where each vertex records a list of adjacent vertices. This simplifies the reverse Dijkstra's search as each vertex not only records its outgoing edges but also its incoming edges.

Pseudocode

```
function read-graph(file) → Graph:
    read number of vertices from file
    read number of edges from file
    graph:Graph ← empty-graph
    initialise graph to have V vertices with infinite shortest-path
    for every edge in file:
        read ‘from vertex’, ‘to vertex’, and ‘weight’ from file
        add edge to from-vertex.outgoing
        add reversed edge to to-vertex.incoming
    return graph

- This function is Dijkstra’s algorithm on the reversed graph
function calculate-heuristic(graph:Graph, destination:Vertex):
    queue:PriorityQueue ← empty-queue
    visited-vertices:Set ← empty-set
    destination.shortest-path ← 0
    queue.push(destination, 0)
    while queue is not empty:
        vertex ← queue.pop()
        if vertex in visited-vertices:
            continue
        add vertex to visited-vertices
        for every incoming edge to vertex:
            if adjacent vertex not in visited vertices:
                path-cost ← vertex.shortest-path + edge.weight
                if adjacent-vertex.shortest-path > path-cost:
                    adjacent-vertex.shortest-path ← path-cost
                    queue.push(adjacent-vertex, path-cost)
```

```
- This function is an A*-search using the shortest path lengths
- calculated in the previous function as the heuristic
function search(graph:Graph, src:Vertex, dest:Vertex, k:Int):
     queue:PriorityQueue ← empty-queue
     - The element pushed onto the queue is the vertex,
     - the current path length (0), and the priority
     queue.push(src, 0, src.shortest-path)
     while queue is not empty:
          (vertex, path-length, _) ← queue.pop()
          if vertex = dest:
               print path length
               k ← k-1
               if k = 0:
                    return
          for every outgoing edge from vertex:
               current-path-length ← path-length + edge.weight
               heuristic ← adjacent-vertex.shortest-path
               priority ← current-path-length + heuristic
               queue.push(adjacent-vertex,
                           current-path-length,
                           priority)
```

Analysis

Results

In the following table, ‘Building Time’ is the time to read in the edge data from the file and build the graph structure. ‘Preprocessing Time’ is the time to perform the reverse Dijkstra’s algorithm and corresponds to the `calculate-heuristic` function. ‘Searching Time’ is the time to find the k shortest paths and corresponds to the `search` function. ‘Total Time’ is sum of the three.

k	Output	Building Time (milliseconds)	Preprocessing Time (milliseconds)	Searching Time (milliseconds)	Total Time (milliseconds)
2	1035.62, 1036.41	15.25	9.68	0.38	25.31
3	1035.62, 1036.41, 1036.45	16.92	7.18	0.28	24.38
5	1035.62, 1036.41, 1036.45, 1036.64, 1036.72	15.04	6.21	0.29	21.54
7	1035.62, 1036.41, 1036.45, 1036.64, 1036.72, 1036.86, 1036.86	20.59	6.74	0.33	27.66
10	1035.62, 1036.41, 1036.45, 1036.64, 1036.72, 1036.86, 1036.86, 1036.95, 1037.15, 1037.19	15.28	6.4	0.3	21.98
20	…	16.75	6.12	0.49	23.36
50	…	15.72	6.66	0.78	23.17
100	…	21.54	6.05	1.38	28.97
200	…	19.27	6.06	2.58	27.9
500	…	14.91	8	6.89	29.8
1000	…	20.46	6.77	11.31	38.54
2000	…	16.27	5.93	21.64	43.85
5000	…	15.93	9.59	60.24	85.76
10000	…	18.99	7.16	109.23	135.38

Output for k > 10 is not given as the list is very long. It is clear that the algorithm is very fast. For k < 2000 the time is largely dominated by the time to simply read in the data from the file and build the graph. The example input graph contains 11825 vertices and 28320 edges, but for k=10,000 the program still only takes about one tenth of a second.

Performance

Building Time

Building the graph takes O(E) iterations of a loop to read in data from a file and iteratively construct the graph structure. In each pass through the loop we must read in the data, construct an edge data structure, and add the index of the new edge to the two vertices to which it is incident. As C++ standard vectors are used to hold all these values, we have constant time random access as well as amortised constant time appends. Therefore each pass through the loop requires amortised constant time, or O(E) in total.

Preprocessing Time

The preprocessing time is more difficult to analyse. Assume that each vertex has approximately the same number of outgoing edges, then if the graph has V vertices and E edges, each vertex has O(E/V) edges. In the `calculate-heuristic` function we perform the loop multiple times until the queue is empty. Each vertex may come off the queue multiple times, but in each instance other than the first, we immediately skip it. Performing this check takes O(log V) times as a C++ set is used.

For each edge incident to the vertex popped off the queue we add that adjacent vertex to the queue. As a C++ priority queue is used, each insertion to the queue takes O(log Q) time where Q is the size of the queue. In each instance we add E/V new vertices to the queue. However, each vertex is processed in this way only once, even if it is added to the queue multiple times. Therefore in the worst case the size of the queue is O(V×E/V), or simply O(E). This can also be seen from the fact that each edge must be considered at most once, but in the worst case, we must consider all of them.

Thus we perform the loop O(E) times, and in each iteration we perform an O(log V) test to see if the vertex has already been visited, so this particular part takes O(E log V) time. However, for all V vertices we must add their adjacent O(E/V) vertices to the queue, taking O(log E) time to do so, or O(E log E) for all vertices. Again, this can also be seen from the fact that in the worst case we consider every edge and add their incident vertices to the queue. This portion takes therefore O(E log E) time total, as this dominates the O(E log V) checking time.

This is not the asymptotically fastest way to compute Dijkstra's algorithm. It is slower because we may add vertices to the queue multiple times even when they should only be processed once, where Dijkstra's original version updates the priority of the elements in the queue in place. However, C++ `std::priority_queue`s do not easily support this. An alternative version of this program was made using a custom binary heap structure to be able to use Dijkstra's original version and achieve O(V + E log V) time, but it was found not to be any faster in practice, so the program was reverted to use this simplified algorithm.

Searching Time

With the exact shortest path length calculated for every vertex in the graph, k shortest paths can be found very quickly. In the beginning of the algorithm we insert a single vertex, the source, into the queue. Then, in a loop, we pop the next vertex off the queue and add its neighbours according to the heuristic priority.

The very first vertex to come off the queue will of course be the source. Every vertex adjacent to it will be inserted the queue, but in a different position according to their priority. In particular, the vertex at the head of the priority queue will be the one next in the shortest path. This is because the priority of any vertex is the cost to reach that vertex plus the heuristic, in other words, the path length from that vertex to the destination. The vertex next along in the shortest path will therefore have the lowest priority.

Suppose the shortest path is 1000, and the cost reach the next shortest is 10. Then the heuristic for that node must be 990, and the total priority will be 1000 again – the shortest path length. Every other vertex will have a greater cost even if they are closer to the source vertex, so their priority will be greater than 1000. In fact, the further we stray from the absolute shortest path, the larger the priority will be.

Vertices will therefore be popped off the queue in the order they appear in the shortest path. In the worst case the shortest path will visit every vertex, therefore we will perform the loop O(V) times before we find the goal. At every pass through the loop we must add that vertex's neighbouring vertices to the queue at a cost of O(log Q) where Q is the size of the queue. If every vertex has E/V neighbours, we will add V×E/V vertices to the queue, so the queue will once again be O(E) in size. The total time to find the goal therefore will be O(V log E).

Of course, this only finds a single path to the goal, where we wish to find k shortest paths. To do this we continue with the A*-search until we find in total k paths. Ideally the next shortest path will share a long prefix with the shortest, in which case finding the next shortest will be only be a few pops off the queue away. However, in the worst case, the next shortest will share no prefix and we will essentially replicate the process again, performing an O(V log E) search k times, or O(k V log E) in total.

In the 'Searching Time' column of the results table it can be seen that the time taken does indeed grow approximately linearly in k, but only for larger values of k, where we must try increasingly greater deviations from the shortest path. For small values of k it is the case that in practice the k shortest paths will share a large prefix with each other and therefore this work is not repeated.

Total

The total time taken by the program is therefore O((E + k V) log E). In terms of space complexity, the size of the graph is dominated by the number of edges, so this is O(E). This O(E) factor is also the space complexity of the priority queues of both the preprocessing and searching phases in the worst case. Therefore the total space complexity is O(E).

Innovation

The primary innovation in this program is its performance, which is very good both in theory and practice, especially when compared to Yen's algorithm. Recall that Yen's algorithm has O(k V (E + V log V)) time complexity. It is difficult to compare them directly, so suppose the graph is complete and therefore contains O(V^2) edges. This program therefore requires O((V^2 + k V) log V), where Yen's takes O(k V^3). Typically k will be much smaller than V so we may treat it as a small constant. In that case this program has time complexity O(V^2 log V), where Yen's has O(V^3).

Not only is this an improvement on paper, it can be seen in the results table that this program is very fast in practice, being able to find the 10,000 shortest paths in the example problem in a fraction of a second. In fact, for any reasonable value of k, the program is dominated in practice by the large constant factor in the O(E) building phase, even though asymptotically it should be the fastest part of the program.

It must be remembered that the O(V log E) time to find a single path in the search phase is the absolute worst case, and it practice will be much faster. This can be seen from the fact that the time to preprocess the graph, taking asymptotically O(E log E) time, requires approximately 7 milliseconds, while the time to find the first shortest path, asymptotically O(V log E), takes approximately only one third of a millisecond.

As k grows larger the searching phase of the program dominates and asymptotic performance becomes like O(k V log E); again, much better than Yen's. This algorithm in particular shows its strength for large values of k, where the performance grows approximately linearly in k.

Improvements could be made to the program by using an asymptotically faster implementation of Dijkstra's algorithm, which would reduce the time complexity to O(E log V + k V log E). However, but this would only be an improvement in the preprocessing phase. For sufficiently large values of k, this becomes irrelevant.